```python
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
import sympy
from sympy import symbols
from itertools import tee
import math
from queue import PriorityQueue
from moviepy.editor import ImageSequenceClip
import time
import cv2
import heapq

# Defining constant values and colors
height = 50
width = 180
linear_threshold = 0.5
angular_threshold = 30
clearance = 2
BACKGROUND_COLOR = (232,215,241)
OBSTACLE_COLOR = (74,48,109)
PATH_COLOR =  (255, 0, 0)
CLOSED_NODE_COLOR = (161,103,165)
OPEN_NODE_COLOR = (56,36,66)
INITIAL_NODE_COLOR = (255,255,255)
GOAL_NODE_COLOR = (0,0,0)
WALL_COLOR = (0,49,83)

# Defining symbols
x,y,z,a,b,r = symbols("x,y,z,a,b,r")

def pairwise(iterable):
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

# class to define lines
class Line():
  def __init__(self,equation,symbol) -> None:
    self.equation = equation
    self.symbol = symbol

  # Function to check point lies on correct side of line
  def check(self,point):
    point = {x:point[0],y:point[1]}
    value = self.equation.xreplace(point)
    if self.symbol == "g":
      return value >= 0
    else:
      return value < 0

# class to define shape
class Shape():
  def __init__(self,lines) -> None:
    self.lines = lines

  #Function to add line to define shape
  def add_line(self,line:Line):
    self.lines.append(line)

  # Function to check whether given point lies inside the shape
  def check_point_inside_shape(self,point):
    verdict = True
    for line in self.lines:
      verdict = verdict and line.check(point)
      if not verdict:
```

```
        break
    return verdict

# class to define shape collection by combining shapes
class ShapeCollection():
  def __init__(self,shapes) -> None:
    self.shapes = shapes

  # Function to add shape
  def add_shape(self,shape:Shape):
    self.shapes.append(shape)

  # Function to check if point lies inside shape collection
  def check_point_inside_shape_collection(self,point):
    verdict = False
    for shape in self.shapes:
      verdict = verdict or shape.check_point_inside_shape(point)
      if verdict:
        break
    return verdict

def get_line_offset(line:Line):
  sign = line.symbol
  if sign == "g": # g for greater than and l for lesser than
    equation = line.equation - clearance
  else:
    equation = line.equation + clearance
  return Line(equation,sign)

# Function to get rectangular shape offset by clearance
def get_shape_offset(shape:Shape):
  return Shape([get_line_offset(li) for li in shape.lines])

#Border
b1 = Line(x-0,"g")  # x = 0
b2 = Line(x-clearance,"l") # x = 2

b3 = Line(x-width+clearance,"g") # x = 178
b4 = Line(x-width,"l") # x = 180

b5 = Line(y-0,"g") # y = 0
b6 = Line(y-clearance,"l") # y = 2

b7 = Line(y-height+clearance,"g") # y = 48
b8 = Line(y-height,"l") # y = 50

# Defining border rectangles
border1 = Shape([b1,b2,b5,b8])
border2 = Shape([b3,b4,b5,b8])
border3 = Shape([b5,b6,b1,b4])
border4 = Shape([b7,b8,b1,b4])
# Adding to shape collection
obstacle = ShapeCollection([border1,border2,border3,border4])

# Defining the upper and lower lines for letters and digits
lower_line = Line(y-10,"g")
upper_line = Line(y-37,"l")
wall_lower_line = get_line_offset(lower_line)
wall_upper_line = get_line_offset(upper_line)

#Letter E

# Defining clearance boundary
line1 = Line(x-20,"g")
line2 = Line(x-29,"l")
e_1 = Shape([line1,line2,lower_line,upper_line])
obstacle = ShapeCollection([e_1])
```

```
line3 = Line(x-37,"l")
line4 = Line(x-20,"g")
line5 = Line(y-28,"g")
e_2 = Shape([upper_line,line3,line4,line5])
obstacle.add_shape(e_2)

line6 = Line(y - 18, "g")
line7 = Line(y - 27,"l")
e_3 = Shape([line3,line4,line6,line7])
obstacle.add_shape(e_3)

line8 = Line(y - 17 ,"l")
e_4 = Shape([line8,lower_line,line3,line4])
obstacle.add_shape(e_4)

# Defining the letter
e_1w = get_shape_offset(e_1)
wall = ShapeCollection([e_1w])
for e_w in [e_2,e_3,e_4]:
  wall.add_shape(get_shape_offset(e_w))

#Letter N
# Defining clearance boundary
line9 = Line(x-39,"g")
line10 = Line(x-48,"l")
n_1 = Shape([line9,line10,lower_line,upper_line])
obstacle.add_shape(n_1)

line11 = Line(x-61,"l")
line12 = Line(x-52,"g")
n_2 = Shape([line11,line12,lower_line,upper_line])
obstacle.add_shape(n_2)

line13 = Line(13*y +27*x - 1534,"g")
line14 = Line(13*y +27*x - 1777,"l")
n_3 = Shape([line13,line14,upper_line,lower_line])
obstacle.add_shape(n_3)

# Defining letter
for n_w in [n_1,n_2]:
  wall.add_shape(get_shape_offset(n_w))

line13w = Line(13*y +27*x - 1594,"g")
line14w = Line(13*y +27*x - 1717,"l")
n_3w = Shape([line13w,line14w,wall_upper_line,wall_lower_line])
wall.add_shape(n_3w)

#Letter P
# Defining clearance boundary
line15 = Line(x-63,"g")
line16 = Line(x-72,"l")
n_4 = Shape([line15,line16,upper_line,lower_line])
obstacle.add_shape(n_4)

line17 = Line(x-70,"g")
line18 = Line((x-70)**2 + (y - 29)**2 - 64,"l")
n_5 = Shape([line17,line18])
obstacle.add_shape(n_5)

# Defing letter
wall.add_shape(get_shape_offset(n_4))
line17w = Line(x-70,"g")
line18w = Line((x-70)**2 + (y - 29)**2 - 36,"l")
n_5w = Shape([line17w,line18w])
wall.add_shape(n_5w)
```

```
# Letter M
# Defining clearance boundary
line19 = Line(x-80,"g")
line20 = Line(x-89,"l")
m_1 = Shape([line19,line20,upper_line,lower_line])
obstacle.add_shape(m_1)

line21 = Line(13*y + 27*x - 2641,"g")
line22 = Line(13*y + 27*x - 2884,"l")
m_2 = Shape([line21,line22,upper_line,lower_line])
obstacle.add_shape(m_2)

line23 = Line(13*y - 27*x + 2327,"l")
line24 = Line(13*y - 27*x + 2570,"g")
m_3 = Shape([line23,line24,upper_line,lower_line])
obstacle.add_shape(m_3)

line25 = Line(x-104,"g")
line26 = Line(x-113,"l")
m_4 = Shape([line25,line26,upper_line,lower_line])
obstacle.add_shape(m_4)

# Defining letter
for m_w in [m_1,m_4]:
  wall.add_shape(get_shape_offset(m_w))

line21w = Line(13*y + 27*x - 2696,"g")
line22w = Line(13*y + 27*x - 2831,"l")
m_2w = Shape([line21w,line22w,wall_upper_line,wall_lower_line])
wall.add_shape(m_2w)

line23w = Line(13*y - 27*x + 2380,"l")
line24w = Line(13*y - 27*x + 2515,"g")
m_3w = Shape([line23w,line24w,wall_upper_line,wall_lower_line])
wall.add_shape(m_3w)

# Digit 6 first

# Defining clearance boundary
line27 = Line((x-126)**2+(y-21)**2-11**2,"l")
s1_1 = Shape([line27])
obstacle.add_shape(s1_1)
line28 = Line((x - 138)**2 + (y-23)**2 - 14**2,"g")
line29 = Line((x - 138)**2 + (y-23)**2 - 23**2,"l")
line30 = Line(y-23,"g")
line31 = Line(x-136,"l")
s1_2 = Shape([line28,line29,line30,line31])
obstacle.add_shape(s1_2)
line32 = Line((x - 137)**2 + (y-41)**2 - 4.5**2,"l")
s1_3 = Shape([line32])
obstacle.add_shape(s1_3)

# Defining digit
line27w = Line((x-126)**2+(y-21)**2-9**2,"l")
s1_1w = Shape([line27w])
wall.add_shape(s1_1w)
line28w = Line((x - 138)**2 + (y-23)**2 - 16**2,"g")
line29w = Line((x - 138)**2 + (y-23)**2 - 21**2,"l")
s1_2w = Shape([line28w,line29w,line30,line31])
wall.add_shape(s1_2w)
line32w = Line((x - 137)**2 + (y-41)**2 - 2.5**2,"l")
s1_3w = Shape([line32w])
wall.add_shape(s1_3w)

# Digit 6 second
# Defining clearance boundary
line33 = Line((x-149)**2+(y-21)**2-11**2,"l")
```

```
s2_1 = Shape([line33])
obstacle.add_shape(s2_1)
line34 = Line((x - 160)**2 + (y-23)**2 - 14**2,"g")
line35 = Line((x - 160)**2 + (y-23)**2 - 23**2,"l")
line36 = Line(y-23,"g")
line37 = Line(x-158,"l")
s2_2 = Shape([line34,line35,line36,line37])
obstacle.add_shape(s2_2)
line38 = Line((x - 159)**2 + (y-41)**2 - 4.5**2,"l")
s2_3 = Shape([line38])
obstacle.add_shape(s2_3)
# Defining digit
line33w = Line((x-149)**2+(y-21)**2-9**2,"l")
s2_1w = Shape([line33w])
wall.add_shape(s2_1w)
line34w = Line((x - 160)**2 + (y-23)**2 - 16**2,"g")
line35w = Line((x - 160)**2 + (y-23)**2 - 21**2,"l")
s2_2w = Shape([line34w,line35w,line36,line37])
wall.add_shape(s2_2w)
line38w= Line((x - 159)**2 + (y-41)**2 - 2.5**2,"l")
s2_3w = Shape([line38w])
wall.add_shape(s2_3w)

# Digit 1
# Defining clearance boundary
line39 = Line(x-165,"g")
line40 = Line(x-174,"l")
line41 = Line(y - 40,"l")
o_1 = Shape([line39,line40,lower_line,line41])
obstacle.add_shape(o_1)
# Defining digit
wall.add_shape(get_shape_offset(o_1))

# Fubction to scale previously made grid
def scale(image,scalex,scaley):
  return np.repeat(np.repeat(image,scalex,axis=1),scaley,axis=0)

# Drawing thr grid

print("Generating the map....")

image = np.full((height,width,3),BACKGROUND_COLOR)
for row in range(height):
  for column in range(width):
    if wall.check_point_inside_shape_collection([column,row]): # Check if point is inside any 
letter/digit
      image[row][column] = WALL_COLOR # color as wall color
    elif obstacle.check_point_inside_shape_collection([column,row]): # Check if point is in 
clearance distance
      image[row][column] = OBSTACLE_COLOR # color as obstacle color
image = scale(image,3,3) # scale the image by 3 times, so clearance of 2 becomes 6
image = np.pad(image,((50-5,50-5),(10-5,50-5),(0,0)),mode="edge") # add padding to get 
required size
image = cv2.copyMakeBorder(image, 5, 5, 5, 5, cv2.BORDER_CONSTANT, value=OBSTACLE_COLOR)

print("Press q to close the window and continue...")

plt.title("Workspace Map")
plt.imshow(image,origin="lower")
plt.show()

frames=[]
begin_time = time.time()

# Canvas dimensions
height, width = 250,600
canvas = np.flipud(image).copy()
```

```python
# Function to flip the y (origin)
def flip_y(y):
    return height - y - 1

# FUnction to check if the coordinate is in free space
def is_free(x, y, canvas):
    if y >= height or x>= width:
      return False
    return np.all(canvas[int(y)][int(x)] == BACKGROUND_COLOR)

# Function to calculate the heuristic (Euclidean distance) between two points
def calc_heuristic(p1, p2):
    return math.hypot(p1[0] - p2[0], p1[1] - p2[1])

# Function to check if the current position has reached the target
def reached_target(current, target, pos_tol=1.5, angle_tol=30):
    dist = calc_heuristic(current, target)
    angle_diff = abs(current[2] - target[2]) % 360
    angle_diff = min(angle_diff, 360 - angle_diff)
    return dist <= pos_tol and angle_diff <= angle_tol

# Function to apply a motion given the current state, angle, and distance (actions)
def apply_motion(current, angle, distance):
    radian = math.radians(angle)
    x_new = round((current[0] + distance * math.cos(radian)) * 2) / 2
    y_new = round((current[1] + distance * math.sin(radian)) * 2) / 2
    return (x_new, y_new, angle), distance

# Function to rotate the robot by an angle offset and move forward by a step size
def rotate_and_move(current, angle_offset, step):
    new_angle = (current[2] + angle_offset) % 360
    return apply_motion(current, new_angle, step)

# Defining the action set
move_options = {
    'FWD': lambda node, o, s: apply_motion(node, o, s),
    'L30': lambda node, o, s: rotate_and_move(node, 30, s),
    'R30': lambda node, o, s: rotate_and_move(node, -30, s),
    'L60': lambda node, o, s: rotate_and_move(node, 60, s),
    'R60': lambda node, o, s: rotate_and_move(node, -60, s)
}

# Function to implement the A* algorithm
def a_star_search(canvas_img, start, target, move_step):

    # Priority queue (open list) initialized with the start node
    open_list = [(calc_heuristic(start, target), start)]

    # Visited nodes
    visited_set = set()

    # Dicts to store path tree and costs
    path_tree = {start: None}
    path_cost = {start: 0}

    space_mask = np.all(canvas_img == BACKGROUND_COLOR, axis=2)
    # Sotring for visualization
    trace_children, trace_parents, visited_nodes = [], [], []

    # Function to check if the point is in free space
    def point_is_valid(x, y, mask):
        flipped_y = flip_y(y)
        if 0 <= x < width and 0 <= flipped_y < height:
            return mask[flipped_y, x]
        return False
```

```
    # Main loop for A* search
    while open_list:
      # Retrieve the node with the lowest cost
        _, node = heapq.heappop(open_list)
        visited_set.add(node[:2])
        next_nodes = []
        # Check if the current node has reached the target
        if reached_target(node, target):
            return trace_route(path_tree, start, node), path_cost[node], visited_nodes,
trace_parents, trace_children

        #  Expand to neighboring nodes
        for _, move_fn in move_options.items():
            neighbor, _ = move_fn(node, node[2], move_step)
            # Skip if the neighbor has already been visited or is not in free space
            if neighbor[:2] in visited_set or not point_is_valid(int(neighbor[0]),
int(neighbor[1]), space_mask):
                continue

            total_cost = path_cost[node] + move_step

            # Update path cost if a better path is found
            if neighbor not in path_cost or total_cost < path_cost[neighbor]:
                path_cost[neighbor] = total_cost
                priority = total_cost + calc_heuristic(neighbor, target)
                heapq.heappush(open_list, (priority, neighbor))
                path_tree[neighbor] = node
                visited_nodes.append(neighbor)
                next_nodes.append(neighbor[:2])

        # If new nodes were added, update trace data for visualization
        if next_nodes:
            trace_parents.append(node[:2])
            trace_children.append(next_nodes)

    return None, None, visited_nodes, trace_parents, trace_children

# fUNCTION to backtrack the path
def trace_route(path_map, origin, endpoint):
    steps = [endpoint]
    # Continue until reaching the origin
    while steps[-1] != origin:
        # Append the parent node
        steps.append(path_map[steps[-1]])
    # Reverse to get the order
    steps.reverse()
    return steps

# Function to visualise the search process
def visualize_path(canvas_img, path, parents, children):
    print("Generating video...")
    # Draw exploration vectors to visualize the search process
    for idx, parent in enumerate(parents):
        for child in children[idx]:
            x1, y1 = int(parent[0]), flip_y(int(parent[1]))
            x2, y2 = int(child[0]), flip_y(int(child[1]))
            #  Draw an arrowed line representing the expansion
            canvas_img = cv2.arrowedLine(canvas_img, (x1, y1), (x2, y2), (200, 160, 40), 1,
tipLength=0.2).copy()

        if idx % 50 == 0:
          frames.append(canvas_img)

     # Draw the final path in red
    for parent,child in pairwise(path):
        x1, y1 = int(parent[0]), flip_y(int(parent[1]))
        x2, y2 = int(child[0]), flip_y(int(child[1]))
```

```
        canvas_img = cv2.arrowedLine(canvas_img, (x1, y1), (x2, y2), (0,0,250), 1,
tipLength=0.2)
        frames.append(canvas_img)

    # Create a video from the frames
    clip = ImageSequenceClip(frames, fps=24)
    clip.write_videofile(f'output_astar.mp4')
    print("Video saved as output_astar.mp4")

# ---- USER INPUT ----

# Start positions
sx, sy, t1 = map(int, input("\nEnter the start coordinates in the form x,y,theta:
").split(','))
while not is_free(sx, sy, canvas) or t1 % 30 != 0:
  print("Invalid start position.")
  sx, sy, t1 = map(int, input("Enter the start coordinates in the form x,y,theta:
").split(','))

# Goal positions
gx, gy, t2 = map(int, input("Enter the goal coordinates in the form x,y,theta: ").split(','))
while not is_free(gx, gy, canvas) or t2 % 30 != 0:
  print("Invalid goal position.")
  gx, gy, t2 = map(int, input("Enter the goal coordinates in the form x,y,theta:
").split(','))

# Step value
step = int(input("Enter the step-size (1-10): "))
while not 1 <= step <= 10:
  print("Step size value should be a value between 1 and 10")
  step = int(input("Enter the step-size: "))

start = (sx, sy, t1)
goal = (gx, gy, t2)

# Perform A*
path, cost, explored, parent_nodes, child_nodes = a_star_search(canvas.copy(), start, goal,
step)

if path:

  print("\nPath found. Total cost:", cost, end="\n\n")
  # Generate animation
  visualize_path(canvas.copy(), path, parent_nodes, child_nodes)

else:
  print("\nNo valid path found.")
```